Master in Artificial Intelligence

Facultat d'Informàtica de Barcelona (UPC)

Facultat de Matemàtiques (UB)

Escola Tècnica Superior d'Enginyeria (URV)

Master Thesis

Glyph

Sebastian Berns

Supervisor

Dario Garcia Gasulla

Co-supervisor

Ulises Cortés

October 2018

Contents

Chapter 1

Introduction

1.1 Problem Definition

The design of a typeface is a manual process in which the type designer shapes each letter, number and symbol by hand. The shapes of a digital typeface's glyphs are defined by points in a bi-dimensional space which connect to straight lines and curves building the glyph's outlines.

For definitions of typography related vocabulary please consult the glossary (appendix B, p. 52)

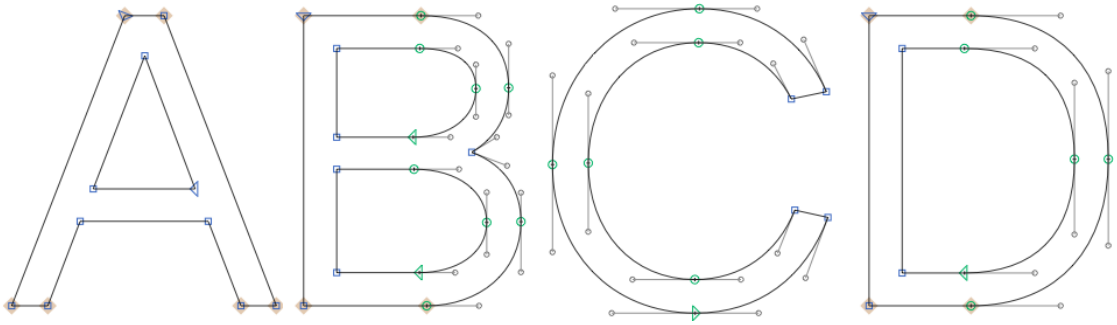


Figure 1.1: Example glyph outlines

While drawing the Latin alphabet's upper and lowercase characters is still fun, extending this character set by Latin characters with modifying diacritics (such as the accents in the Romance languages) is an additional effort. To then move on to other alphabets or scripts, such as Greek or Cyrillic can be quite tedious. Extending the design of a typeface to Simplified Chinese with its over 27,000 logograms not only requires great expertise but can also seem an almost impossible endeavor.

However, there is a great demand for typefaces with support for underrepresented languages as the population of more and more countries finally gain access to personal computers and

the web and (especially digital) companies strive to communicate visually consistently in any human language. Google joined forced with the biggest company in typeface design, Monotype, to develop Noto, which "spans more than 100 writing systems, 800 languages, and hundreds of thousands of characters."[1] Likewise, Airbnb is proud of its custom typeface Cereal designed by the digital type foundry Dalton Maag: "We currently support seven non-Latin based languages, and are looking to internationalize far beyond that."[2] Looking at these examples the dimension of such an undertaking becomes clear. Noto has taken only five years in development thanks to a long list of collaborators, both at Monotype and external.

It is on this ground that I proposed the project which I am here presenting. My original intention thus was to able to generate a complete character set from only a few letters of a particular style. For example, to obtain the upper case letters of a font from its lower case letters, or the characters of the Arabic script from the Cyrillic script.

The challenge here lies in distinguishing the shaping influence of two concepts: content and style. In the particular case of letterforms the question is which parts of a glyph's shape convey the meaning of a letter, and are necessary so that a glyph depicts the letter it is supposed to represent. And which parts of its shape are an expression of its particular stylistic character, and are therefore interchangeable with other characteristics.

If we think about the differentiation of content and style we can find parallels in other media. In speech, for example, the same phonemes are spoken in different ways by different people due to particular dialects and accents. Here for a human the differentiation of style and content is easy: we are able to imitate others' accents while maintaining the meaning of what we say. If we go one step further and think about the semantics of expression, we can risk drawing a parallel even to text and literature. Every author has their particular style of writing and there are different registers in language used for particular purposes or social settings. That is to say, there are various ways to express the same message. Dividing style and content in this context means to be able to change *how* something is said without changing *what* is said.

In the following section 1.2 I have reviewed previous proposals to similar and related problems. In chapter 2 I draft a basic approach to my problem and formulate the objective of this project.

[1] https://www.monotype.com/resources/case-studies/more-than-800-languages-in-a-single-typeface-creating-noto-for-google/ (last retrieved October 3rd 2018)

[2] https://airbnb.design/introducing-airbnb-cereal/ (last retrieved October 3rd 2018)

1.2 Related Work

One of the earliest approaches to "Separating Style and Content" goes back to 1996 [1]. Tenenbaum and Freeman propose a solution based on bilinear factorization for extrapolation of a given style to new content, classification of observed content in a new style and translation of new content in a new style (see figure 1.2).

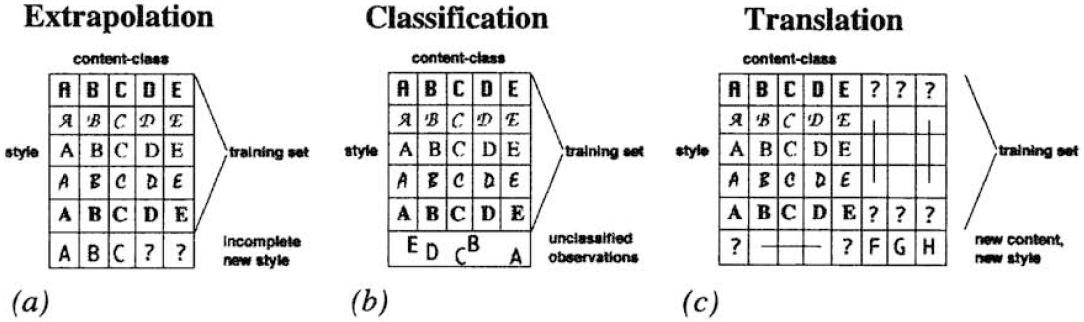


Figure 1: Given observations of content (letters) in different styles (fonts), we want to extrapolate, classify, and translate observations from a new style or content class.

Figure 1.2: Graphic from Tenenbaum and Freeman [1]

A similar formalization, albeit strictly conceptual, can be found in a response by Douglas Hofstadter [2] to the proposal of Metafont by Donald Knuth in 1982. As here reproduced in figure 1.3 Hofstadter places different fonts in separate rows such that their letters line up to form separate columns and formulates two questions: "What do all the items in any column have in common?" and "What do all the items in any row have in common?" That is to say, what do all the 'a's and

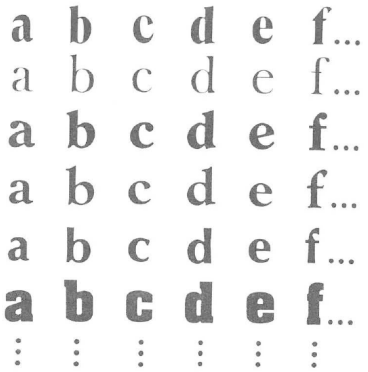


Figure 13. The vertical and horizontal problems.

Figure 1.3: Graphic from Hofstadter [2]

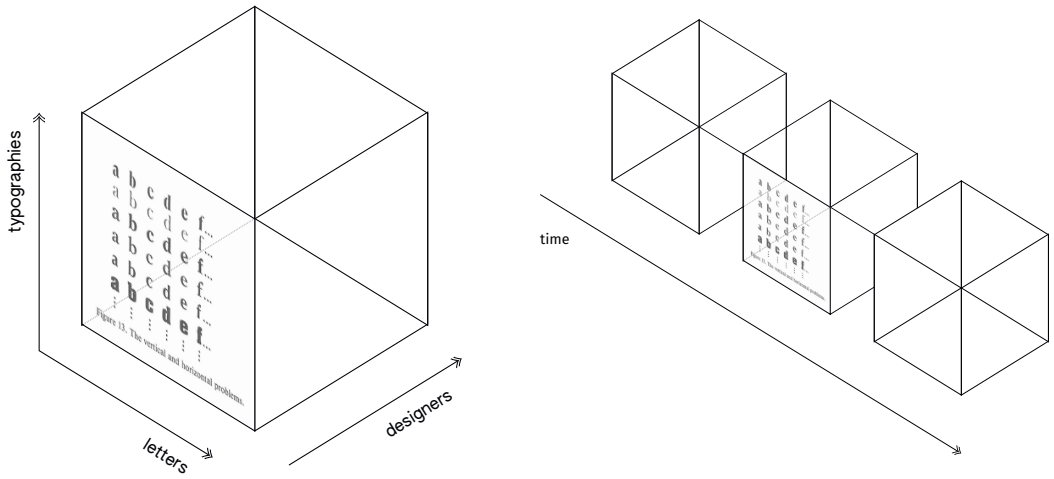


Figure 1.4: Extending Hofstadter's idea to three and four dimensions

'b's in common that makes them represent that very concept of a letter? And similarly, what is the essence of the style of a typeface which unites all its glyphs and causes their internal stylistic consistency. He then goes a step further and adds a third dimension (figure 1.4) to form blocks which group all the typefaces designed by the same person, and asks the same question: what do all these have in common? Hofstadter is of course not shy to extend his *Gedankenexperiment* by the dimension of time. If we were to place these block of designer-specific typefaces chronologically along the years, what do the different stylistic epochs have in common?

Kunth's Metafont [3] is technically interesting for the given problem as it implements a stroke and pen representation for the glyphs of digital typefaces which already achieves a partial division of style and content. The basic shape of a glyph is defined by a skeleton stroke, along which a modifiable pen is moved. The pen can have different base shapes, can be rotated, tilted and skewed. The glyph's final shape is ultimately computed from the dynamic of this parametric system as it moves along a path, which is very much akin to manual calligraphy and the movement of the nib.

I have not found any attempt that made use of the Metafont glyph representation, but it would be interesting to see. Other approaches, however, have developed similar but custom representations on the same notion of strokes and pens [4], or skeletons and parts [5]. These techniques, though, feel overly complicated, need intensive pre-processing and still only produce mediocre results. They do may make sense for Japanese and Chinese characters which are very close to their calligraphic originals and could benefit from this approach.

There are two publications in deep learning that have the highest relevance to my proposal,

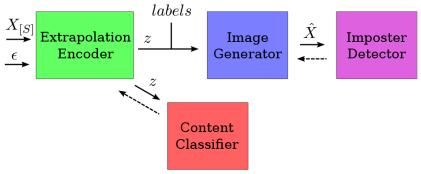


Figure 2. **Model overview.** The variational encoder (green) takes a subset of the style set images $X_{[S]}$ (example shown) and a noise variable $\epsilon \sim \mathcal{N}(0, I)$ as input, and produces code z for all members of the style set, and combines the codes with identity class labels. The generator (blue) to image reconstructions $\hat{X}$ (example shown) for the entire style set. An imposter detector adversary (purple) is used to improve the reconstruction quality. An identity classifier adversary (red) is used to promote class invariance on z codes. Dotted lines indicate zones which update weights with a negated gradient wrt loss functions in the adversarial zones.

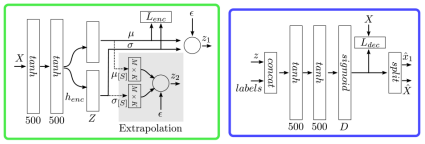


Figure 4. **Extrapolation encoder and image generator. Left**: The extrapolation encoder produces two sets of latent distributions. One set (z_1) corresponds to a standard VAE encoder. The second set (z_2) is produced by our extrapolation layer (shaded portion). See Sections 5.1, 5.2 for details. **Right**: The image generator produces images for both sets of codes. All generated images are compared against ground truth. The z_2 images are the analogical reasoning output. See Section 5.3.

Figure 1.5: Network architecture by Upchurch, Snavely, Bala [6]

and that have both produced good results. Upchurch, Snavely and Bala [6] frame the problem of separating style and content as a single-image analogy task, which consists in learning the necessary prior over either style or content in order to be able to generate a same-styled letter B given only a single letter A. They propose an elaborate neural network architecture (figure 1.5), similar to a generative adversarial network, around a modified variational autoencoder.

Baluja [7] takes a straight-forward approach and proposes the use of deep neural networks, both with convolutional layers and fully connected only, that are capable of learning two tasks from a small subset of only four letters (B, A, S, Q). First, to discriminate whether a fifth letter belongs to the same font or not, and second, given these four base letters to generate the rest of letters in the same style. The only justification of his choice of letters, however, is the intuition that "they contained a diverse set of edges, angles and curves that can be rearranged/analyzed to reveal hints for composing many of the remaining letters." Solving the classification task guides his search in finding adequate network architectures, which can then be promising candidates for the generation task. As quantitative measure for the produced letters he employs the previously trained discrimination models to assess whether the output of the generative networks integrates well into the respective font.

Fig. 6. Single Vs. Multi-Letter Generation. The left networks show individual letter creation (only 2 of 22 shown). The right network generates all letters simultaneously, thereby allowing the hidden units to share extracted information. The tower/column architectures are employed to transform the basis letters (as was done for the discrimination task). The generative network also *auto-encodes* the basis letters (Right).

Figure 1.6: Network architecture of Baluya [7]

Chapter 2

Objective

Having reviewed the available literature it becomes evident that generating the complete set of uppercase letters from only a few characters has been achieved. I therefore want to focus on another approach based on a method from deep learning called Style Transfer. Gatys et al. [8] have successfully used the VGG image classifier model [9] to modify the pixels of a photograph in such a way that it adapts the stylistic characteristics of another image while maintaining the originally pictured content.

Figure 2.1: Example of original style transfer. From the content image (left) and the style image (Vincent van Gogh's *The Starry Night*, 1889, small in the middle) generate an image of the content in the corresponding style (right)

We can translate this principle to the given problem, in which we essentially want to apply a specific typographic style to a set of letters without changing theirs semantics, which is to say the letter they depict, or their content. However, instead of using a pre-trained model such as VGG, I will develop and train a custom classifier architecture, in order to obtain a base model for the Typographic Style Transfer.

I have chosen to limit the project's scope and thus simplify the problem in the three aspects. Firstly, I will work with pixel data. While it would be desirable from the perspective of a type designer to directly generate the scalable vector outlines of a glyph, the outline representation is much more complex from the perspective of the algorithm. The abstract list of points with

bi-dimensional coordinates lacks the spatial context which is intrinsic to pixel images. Since a deep neural network is at the core of my approach, I can benefit from the good performance of convolutions in image recognition tasks. Secondly, only uppercase letters of the Latin alphabet will be taken into consideration. Although it is specifically interesting to generate infrequently used characters (from the western point of view: e.g. the Brahmic script used in the Burmese alphabet) from the most common ones (in this case: the Latin alphabet), data on these digitally underrepresented scripts and languages is sparse. Furthermore, many freely accessible typefaces lack the Latin lowercase letters and consequently the uppercase letters are the most widely available character set. And finally, the problem is reduced to transferring the style of only one glyph to another at a time.

Figure 2.2: Illustration of algorithm inputs and expected output

In summary, the objective is to develop an algorithm that takes as input one content glyph image, defining which letter to generate, and one style glyph image, specifying the desired typographic style, such that the algorithm returns a transfer image depicting the letter from the content image in the style of the style glyph. The methodology of this Typographic Style Transfer, as illustrated in figure 2.2, is explained in detail in chapter 5.

For this style transfer to work we need a custom classifier which given two glyph images is able to predict at a high accuracy if the two are of the same typographic style or not. The development and training of this classifier model is documented in chapter 4. The source and the preparation of the data set necessary for the training and evaluation is laid out in the following chapter 3.

Chapter 3

Data Set

The data set used in this project has been kindly made available by Azadi et al. as collected for their MC-GAN [10]. It had been build from a total of 12,155 fonts, downloaded from the web, separated into three sets:

1. *train* (9,122 fonts)
2. *val* (1,473 fonts)
3. *test* (1,560 fonts)

Each font comprises the capital letters A–Z and has thus a total of 26 glyphs.

All outline font files (True Type and Open Type Fonts) had been rendered to pixels in a rasterization step where each glyph is placed in black on a white background and is then scaled and positioned to fit a bounding box of 64×64 pixels at its maximum size. The final data has one greyscale channel with pixel intensity values ranging from 0 (black) to 255 (white).

Figure 3.1: Examples for the variety of fonts in the *train* set. The uppercase letters A–Z are the complete character set used in this project.

In all experiments of this project the *train* set font have been used exclusively for the training of models, *val* for their evaluation and the tuning of hyperparameters, and the fonts in *test* only to generate new glyphs in the typographic style transfer.

3.1 Preprocessing

Before feeding glyph data to a network it is preprocessed in the following two steps. First the glyph images are inverted such that we obtain a white letter on a black background. That is to say that white is remapped to the value 0 and black to the value 255, as well as all intermediate values correspondingly. This inversion ensures that we consistently maintain the information of interest on a uniform background, since the padding operation of convolutional layers in neural networks adds zeros all around the convolution's output. Finally, the pixel intensity values are normalized to the range [0, 1].

3.2 Permutations

Let F be the number of fonts, let L be the number of letters. Let any sample be composed of two different glyphs. The first can be any of the L letters, while the second has to be picked from the remaining $L-1$ possibilities. Thus, there are $L \times (L-1)$ possible combinations.

In a positive sample both glyphs have to be from the same font, which we can pick from the F possibilities. Negative samples combine two glyphs from different fonts. The first is any of the F fonts, while the other must be any other of the remaining $F-1$. Thus, we count a total of $F \times (F-1)$ possible negative combinations.

Putting together the choices of letters and fonts, we can formalize the total numbers of samples in the sets of positive samples $\mathcal{P}$ and negative samples $\mathcal{N}$ as follow.

$$|\mathcal{P}| = L \times (L-1) \times F$$
$$|\mathcal{N}| = L \times (L-1) \times F \times (F-1)$$

We thus can effectively generate $F-1$ as many negative samples as positive ones. In a balanced data set $\mathcal{B}$, however, the total number of samples is limited by the lower bound of positive samples.

$$|\mathcal{B}| = 2 \times |\mathcal{P}| = 2 \times L \times (L-1) \times F$$

3.3 Training Set

In the case of the given *train* set of $F = 9122$ fonts and $L = 26$ letters, the total number of samples of a balanced training set $\mathcal{T}$ would amount to almost 12 million.

$$|\mathcal{T}| = 2 \times 26 \times (26 - 1) \times 9122 = 11\,858\,600$$

Generating and storing the entirety of these samples is unfeasible. Instead we build only a small subset at each iteration as described below.

Given a set of fonts F and a batch size N, at each iteration:

1. Shuffle the set of fonts.
2. Retrieve a batch B of $\frac{N}{2}$ fonts from the set.
3. For each font f in the batch generate:
    - one positive sample by randomly selecting two different glyphs from the same font.
    - one negative sample by randomly selecting a second different font from the batch $B - f$. From either of the two fonts randomly select a glyph, such that these are not instances of the same letter.

We obtain a balanced batch of N training samples. Although at each iteration we only sample a subset of fonts, which we use to compose random combinations of glyphs, over the course of multiple epochs this approach will generate an increasingly larger part of the possible permutations and only few repetitions.

3.4 Evaluation Set

A balanced evaluation set would have a maximum number of $|\mathcal{E}| = 2 \times 26 \times (26 - 1) \times 1473 = 1\,914\,900$ samples. Not all possible permutations are needed, however. We will limit the size of the evaluation set to 10,000 such that it contains at least 6 positive and negative samples for each font. In order to be able to compute reliable metrics across different epochs we will build a fixed set once and evaluate on the same at each iteration.

1. From the set of fonts F in the *val* set randomly select a font f and generate:
    - one positive sample by randomly selecting two different glyphs from the same font.

- one negative sample by randomly selecting a second different font from the batch $F - f$. From either of the two fonts randomly select a glyph, such that these are not instances of the same letter.

2. Repeat until the desired number of samples has been generated.

Chapter 4

Classifier

The classifier network's architecture is made up of two main parts (see figure 4.1). A primary set of convolutional layers is used for feature extraction (figure 4.2). The following fully connected layers then act as discrimination network in the classification task (figure 4.3).

The two input glyph images are not handled separately but passed through the same convolutional layers. We could therefore say that the convolutional layers share their weights on both inputs. If we treated the two input images separately in two different columns of convolutions

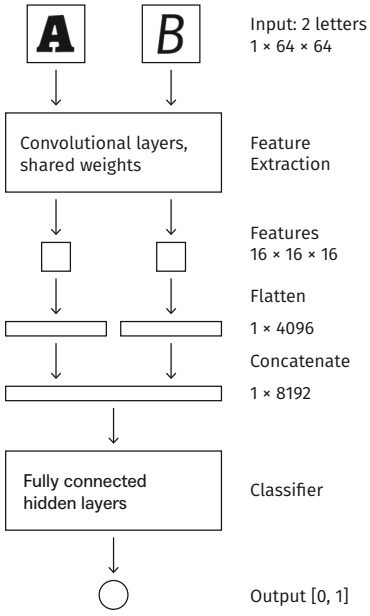


Figure 4.1: Data flow in the model's architecture

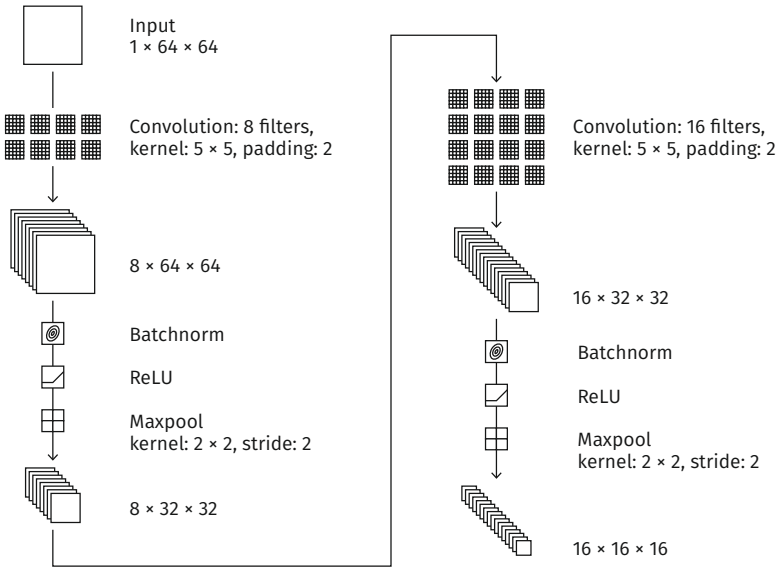


Figure 4.2: Feature extraction

the filters might develop differently and not compute equal features for the same image. The intuition is thus that we want to ensure symmetry in the network and independence of the position of an input image.

The set of convolutions consists of two layers, the first comprising 8 filters and the second 16 filters. Both convolutional layers have a kernel size of 5×5 and are set to add 2 pixels of padding all around, such that they maintain the dimensions of the input matrices. Either one is followed by a Batchnorm layer [11]. The output is then passed through a ReLU and finally on to a max pooling layer with a kernel size of 2×2 and a stride of 2. This way the width and height of the inputs is halved two times while their depth increases.

The features extracted from the two input images thus have a final depth of 16 channels and a height and weight of 16 pixels each. Before being passed on they are prepared for the fully connected layers. First, both matrices are flattened by placing all of theirs rows next to each other such that they become vectors with 4,096 entries each ($16 \times 16 \times 16$). Then the two feature vectors are joined, appending one to the other, to form a composite feature vector of size 8,192.

The combined features are then passed through two fully connected hidden layers, of 2,048 and 512 units respectively and a ReLU activation function. The output layer has one single unit and is activated by a Sigmoid function.

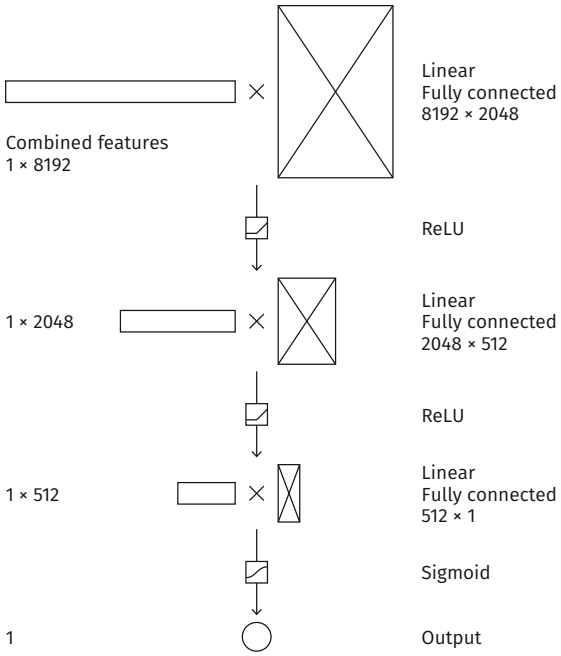


Figure 4.3: Fully connected hidden layers and output layer

4.1 Cross-Entropy Loss

This loss function, also known as log loss, measures the error of a classification model that returns probability values between 0 and 1.

$$L_{\mathrm{BCE}}(p, y) = -(y \times \log p + (1 - y) \times \log(1 - p))$$

where y is the target, and p is the predicted probability [0, 1] as put out by the model.

In the binary case the target is either true (1) or false (0). The loss function for both cases is here shown in figure D. The cross-entropy loss increases as the probability prediction p diverges from the target value y. While the loss only grows slowly, almost linearly, up until a difference of $|y - p| \leq 0.7$, it starts to grow exponentially from there on. Cross-entropy loss thus especially penalizes wrong, but confident predictions.

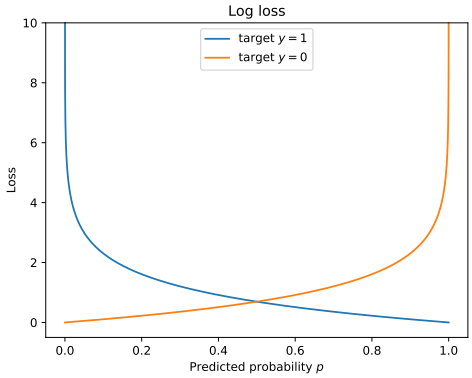


Figure 4.4: The log loss function in the binary case

4.2 Optimizer

To find a well performing combination of optimization algorithm and hyperparameters, I have conducted a few short selective experiments on the real data set and final classifier architecture. First, I separately tested different configurations of algorithms and their hyperparameters or their variations:

- Stochastic gradient descent without and with Momentum (0.1, 0.5, 0.9, 0.95, 0.99) [12]
- Adadelta with hyperparameter ρ (0.9, 0.95, 0.99) [13]
- Adam and its variants AMSGrad and Adamax [14] [15]

- Adagrad [16]

These first experiments have been run over 100 epochs. To make all comparable, learning rates of Momentum and Adadelta have been set to 0.01, and to 0.001 for Adam and Adagrad. From these four comparisons I selected the best candidates and let them compete against each other in a run over 1000 epochs with the earlier defined candidate-specific learning rates.

All configurations in a test have been assigned a separate model of the same architecture. But all of these models have been initiated with the same random weights and have been fed the same training data at each iteration.

SGD/Momentum

In the given problem a higher Momentum in SGD helps in reducing the validation error quickly (see figure 4.5). However a maximum parameter value does not equal best performance. While the curves corresponding to the values 0.9, 0.95 and 0.99 are very close, overlapping each other, Momentum 0.95 performs best out of all six configurations.

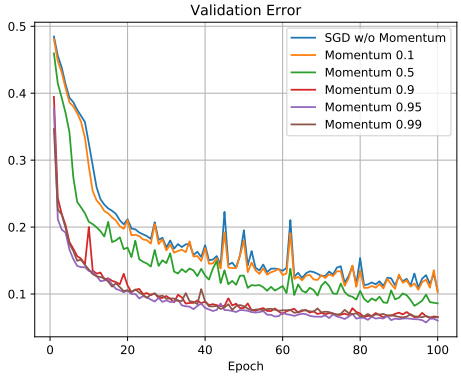


Figure 4.5: Validation error using stochastic gradient descent with and without Momentum

Adadelta

In the Adadelta algorithm ρ is a "decay constant similar to that used in the momentum method" [13]. We can observe in figure F that a higher value for ρ contributes to bigger oscillations in validation error. With $\rho = 0.99$ (green curves) Adagrad on the one hand continuously makes the biggest steps towards the minimum error, but can on the other hand easily overshoot and be outperformed by the other two candidate configurations ($\rho = 0.9$ and $\rho = 0.95$) whenever the validation error again increases temporarily (note spikes around epoch 50 and 95). In the long run, however, Adagrad with $\rho = 0.99$ can be still considered the best candidate.

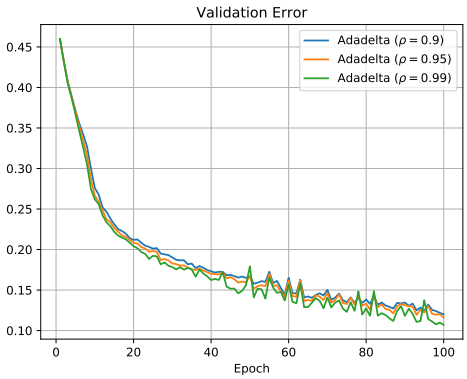


Figure 4.6: Validation error using Adadelta in different configurations

Adam

All three candidate variants perform equally well, and especially fast, considering the rapid descend in the first 20 epochs. Although all three curves are very close and sometimes overlap each other, Adamax is the worst performing of the three. Adam and AMSGrad are indistinguishably close during the first 20 epochs, after which AMSGrad takes the lead in performance, for which it is selected as the best candidate.

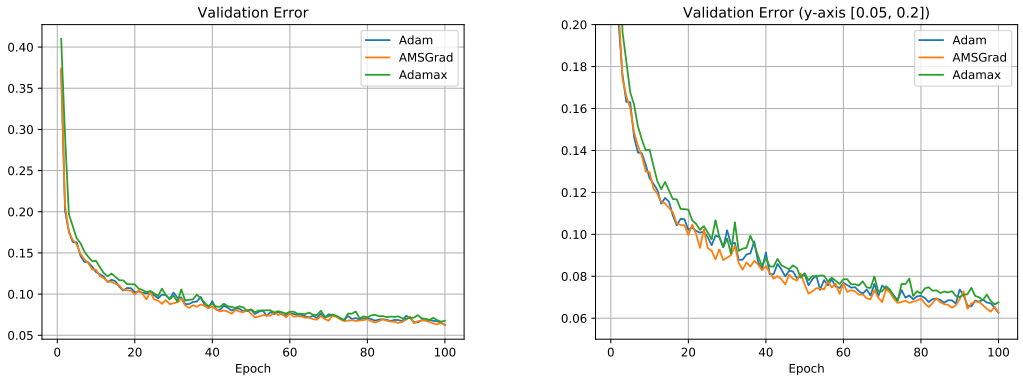


Figure 4.7: Validation error using different variants of the Adam algorithm, right: amplified

Best Candidates

The four candidate optimization algorithms competing in the final comparison are Momentum 0.95 (learning rate 0.01), Adadelta 0.99 (learning rate 0.01), AMSGrad and Adagrad (both learning rate 0.001). After 1000 epochs of training Momentum is leading (97.44 % evaluation

accuracy), but with merely one percentage ahead of AMSGrad (96.48 %). Nevertheless, over the complete period of epochs, as it is visible in figure 4.8, the gap between Momentum and AMSGrad has continuously been growing bigger, and would probably keep growing with further epochs of training.

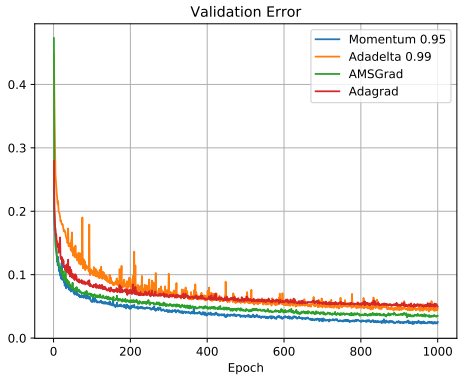


Figure 4.8: Validation error of final candidate configurations

4.3 Training

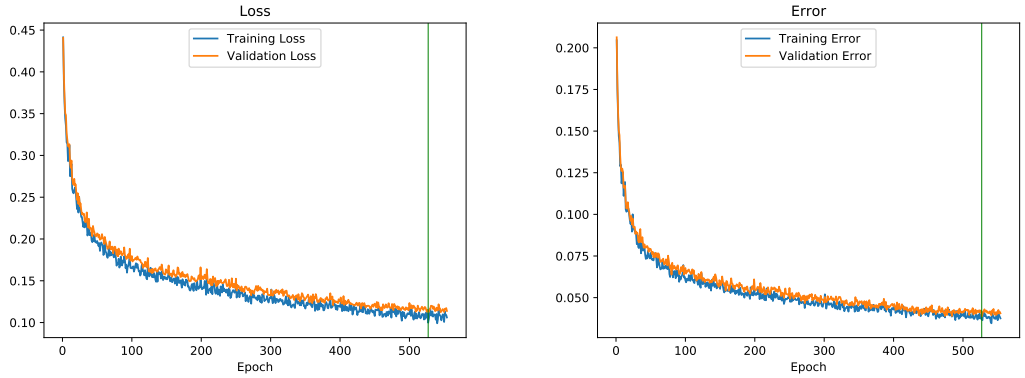


Figure 4.9: Training and validation losses and errors, the highest accuracy is marked in green (epoch 527)

The model was finally trained with stochastic gradient descend and Momentum 0.95 at a learning rate of 0.01 over 550 epochs. During all these iterations the model has processed a total of 10,034,200 positive and negative examples. Both graphics in figure I show the progressive minimization of the training and validation losses, as well as the continuous reduction of the classification error over both the training and the validation set.

While there is surely room for further improvement, with this simple architecture the model already achieved an accuracy of 96.13 % on the validation set at epoch 527 (marked green in figure 4.9). Increasing the network's capacity, e.g. adding more filters to the convolutions or adding more convolutional layers, and also training the model for an increased number of epochs could bring even better performance in the given classification task. However, since the emphasis of this project lies on the subsequent transfer problem, I focused on employing the model at this state in the Typographic Style Transfer as explained in the following chapter 5.

Chapter 5

Typographic Style Transfer

The original style transfer, developed by Gatys et al. [8], is based on the observation that in a deep convolutional neural network it is possible to reconstruct the input from only the response of a particular layer. They have devised a method that also reconstructs the texture information

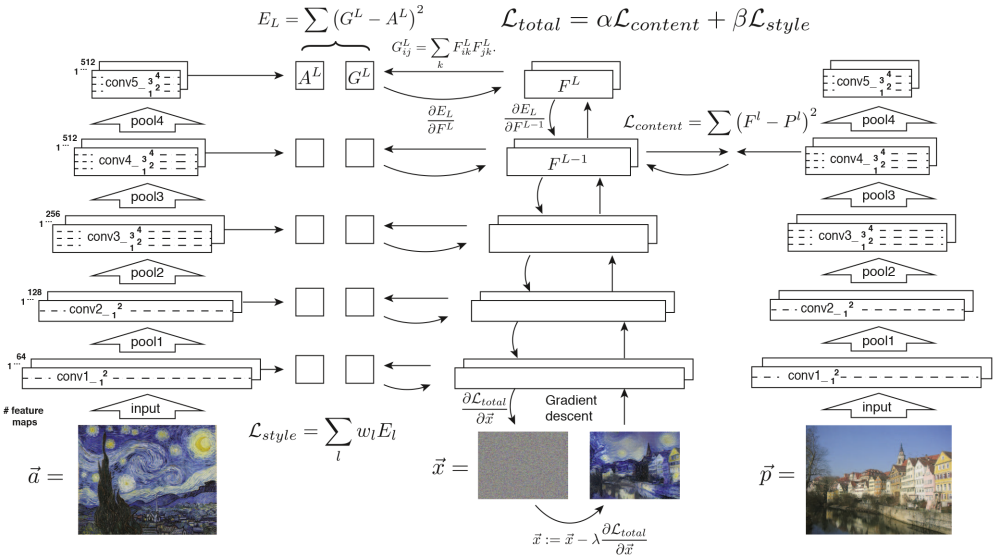


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image $\vec{a}$ is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image $\vec{p}$ is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image $\vec{x}$ is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss $\mathcal{L}_{style}$ (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss $\mathcal{L}_{total}$ is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image $\vec{x}$ until it simultaneously matches the style features of the style image $\vec{a}$ and the content features of the content image $\vec{p}$ (middle, bottom).

Figure 5.1: Elaborate description of the algorithm by Gatys et al. [8]

of an input image by computing the correlations between different features in different layers. With this method it is possible intercept a network's internal representations of image style and content at any layer and to modify an input image in a such a way that its network responses match those of any given reference images for both style and content. The authors have shown impressive results in changing the style of a photo, while maintaining its content, to that of, for example, an expressionist painting.

In contrast to this original method, which is build on the pre-trained VGG image classifier [9], in this project I am pursuing an approach based on a custom classifier model (see chapter 4). In the given context this model has been trained to determine whether two glyphs belong to the same font or not. It is to be expected that the convolutional layers of the model have learned a representation of glyph styles, which serve the fully connected layers as features in the discrimination task. The basic idea is to use this internal representation as a guide for the pixel modification of one of the input glyph images, such that it adapts the style of the other. For this, the network weights have to be fixed and an optimization has to be performed only on the pixel values of one input image according to the model's response. This procedure is described in detail in the following section 5.1.

5.1 Optimization

Let the algorithm take as input a tuple of glyph images: (1) a style image, and (2) a content image. Based on these we want to generate a glyph image that shows the same character as the content image, but in the typographic style of the style image. Let's call this 1–1 transfer.

Figure 5.2: Content image, style image and target image (ground truth, and expected transfer result) (from left to right)

On each transfer step the content image is modified in order to become gradually more similar in style to the style image. Let's call this modified image transfer image, as it is the subject of the style transfer. Note that the transfer image is equal to the content image before the first iteration only. The following steps describe how to perform gradient descent on the transfer image.

1. Feed both the style image and the transfer image as inputs to the classifier model (with fixed weights).
2. Measure the predicted probability p that both input glyph images have the same style, as put out by the model.

3. Compute the loss term (given the prediction p and target y).
4. Compute the gradient of the loss with respect to the transfer image.
5. Update the transfer image's pixel values in correspondence to the optimization algorithm's calculations based on the gradient and learning rate.
6. Limit the transfer image's pixel values to [0, 1].

5.2 Basic Loss Term

Just as during the training of the classifier model the loss is calculated as the cross-entropy (section 4.1) between the prediction p and the target. This time the target is fixed to value of 1, which is equivalent to both input glyphs having the same typographic style. Let's call this the Similarity Loss. The loss term thus enforces the modification of the target image such that the model's output probability gradually increases towards the objective.

$$L = L_{\text{sim}}(p) = L_{\text{BCE}}(p, 1)$$

5.3 Metrics

On the one hand we can do a qualitative evaluation of the output of a typographic style transfer simply by assessing whether the generated glyph visually integrates well into the style font's set of glyphs or not. On the other hand we're able to obtain a quantitative measure if we let the classification model do the same task, since it was specifically trained to estimate the probability of similarity in style of two glyph images. As neither of the two is a sufficient criterion by itself we will rely on both.

Classifying a generated glyph against the whole set of glyphs of the style font, thus, gives us a measure of how well the transfer image has been adjusted to the style of the style font. Similarly, we can measure the opposite: classifying a generated glyph against the content font determines how close the stylistic characteristics of the generated glyph still are to the content font.

These two metrics, the style and content font classification scores, give an insight into how well the glyphs of a font can be generated from the shape of another font's glyphs.

Baseline

The stylistic similarity between two fonts, of course, can be higher or lower from the beginning. A bold and a light cut from the same typeface family do have more in common, albeit their difference in weight, than a blackletter and a handwritten font. It is reasonable to assume that given two fonts it is easier to generate the glyphs of one from the other's the closer both are stylistically. Measuring the mean classification score of all letter combinations between two fonts gives a baseline measure of stylistic similarity (see the next section 5.4). Furthermore, we can measure the similarity of a font to itself to get an idea about the font's internal stylistic consistency. This establishes a baseline to aim for when generating glyphs in the typographic style transfer.

5.4 Design of Experiments

(a) Deja Vu Sans Bold

(b) Deja Vu Serif Bold

(c) Sebaldus Gotisch

(d) Miniquest

Figure 5.3: Fonts from the *test* set

From all the possible transfer tests between two fonts, I have decided to choose problems that serve as clear indicators of success or failure of an experiment. A combination of two font from the *test* data set that can be considered such a problem are Deja Vu Sans Bold and Deja Vu Serif Bold (figure 5.3a and 5.3b). They belong to the same typeface family, have a similar weight, and the important difference between the two is that one is a serif and the other a sans-serif typeface. The algorithm is supposed to accomplish to remove the serifs from a glyph when doing a transfer from Serif to Sans, or add them when doing a transfer in the other direction.

A more difficult combination would pose a typographic style transfer between the fonts Sebaldus Gotisch and Miniquest (figure 5.3c and 5.3d), due to their big differences in characteristics. Here

the algorithm has to do a much bigger effort than just adding or removing serifs.

The aforementioned baseline metrics (section 5.3) help to measure quantitatively what I have described in the previous paragraphs. In table 5.1 we can observe how formal fonts have a high internal stylistic consistency, while the informal Miniquest achieves a lower score in auto-classification. Likewise the two fonts from the typeface Deja Vu have the highest similarity.

	Deja Vu Sans Bold	Deja Vu Serif Bold	Sebaldus Gotisch	Miniquest
Deja Vu Sans Bold	0.9517	0.1258	0.0252	0.0037
Deja Vu Serif Bold	0.1287	0.9798	0.0313	0.0031
Sebaldus Gotisch	0.0115	0.0152	0.9555	0.0084
Miniquest	0.0014	0.0048	0.0143	0.7663

Table 5.1: Baseline font classification scores. Note that the scores are not symmetric. They differ depending on the position a fonts' glyphs was fed to the classifier (either left or right, see model architecture figure 4.1 in chapter 4).

Chapter 6

Experiments

I want to begin this chapter with an important observation about the influence of the optimization algorithm on transfer result in section 6.1.

Then, starting with the basic approach described in the previous chapter 5, I will show the first results of experiments performed with the proposed methodology in section 6.2. The analysis of these first results has naturally lead to further questions and experiments that are summarized along the rest of this chapter (sections 6.3 to 6.5).

The primary experiment consists in the sans-to-serif transfer with the fonts Deja Vu Sans Bold and Deja Vu Serif Bold (see section 5.4). I have done qualitative assessments of the transfer results on selected letter combinations (e.g. T–L, figure 6.1).

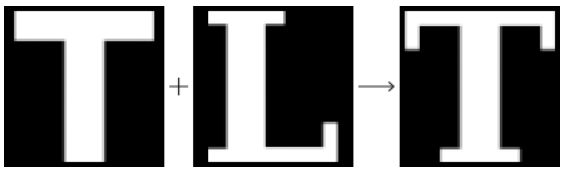


Figure 6.1: Example for sans-to-serif transfer from T to L. From left to right: content image, style image and expected result.

If not indicated otherwise, in all experiments the complete set of transfer glyphs have been generated from all possible combinations of content and style glyphs. That is to say, given a glyph of the letter A from the content font and a glyph of the letter B from the style font the algorithm produces a transfer glyph that shows letter A in the style of B. Both A and B can be any of the 26 uppercase letters (A–Z). The total number of possible permutations therefore is $26 \times 26 = 676$. The reported metrics (loss and classification scores) are averages of the results corresponding to all 676 generated transfer glyphs.

6.1 Optimization Algorithms

While I had previously compared different optimization algorithm configurations for the training of the classification model, I only focused on the most promising candidates (see section 4.2) for the style transfer task. Note that AMSGrad has been omitted from the following analysis, since it showed results identical to Adam. The following comparison is based on the sans-to-serif experiment described in the beginning of this chapter 6.

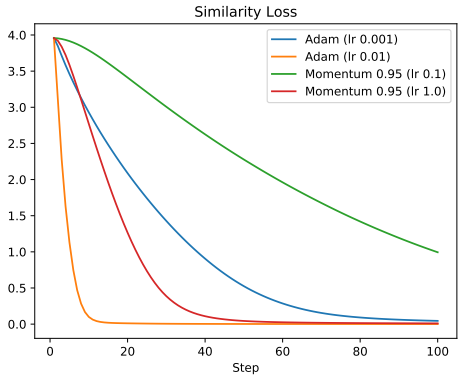
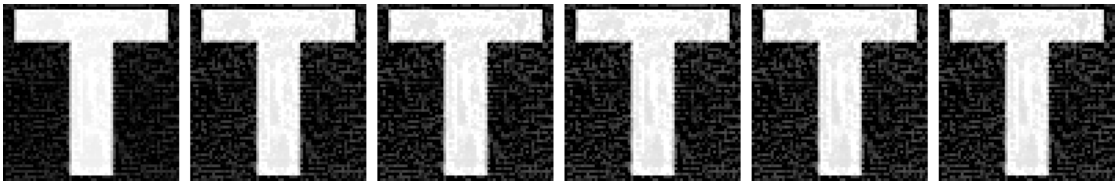


Figure 6.2: performance of different optimization algorithm configurations

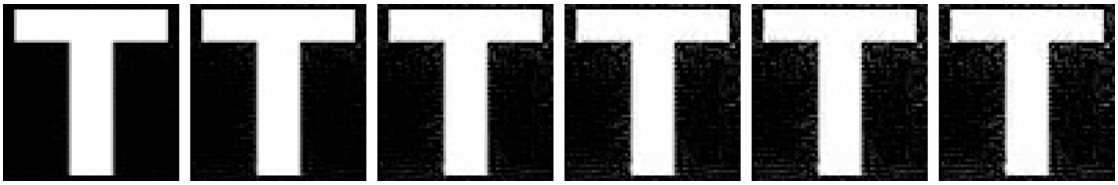
Figure 6.2 compares the optimization performance on the basic loss term (see section 5.2) of Adam and SGD with Momentum 0.95 at different learning rates. The first thing to notice is the remarkable influence of the learning rate. Not only do the two algorithms perform best with two very different values, but changing the learning rate also causes an immense change in the performance of either.

While Adam with a learning rate of 0.01 (orange curve) is the fastest to converge, Momentum 0.95 with a learning rate of 1.0 (red curve) approaches convergence much more smoothly. The effect this has on the actual transfer image is visible in figure Y. As Adam rushes to reduce the loss it quickly introduces and enforces a lot of noisy patches. Momentum meanwhile modifies considerably less pixels. At step 60 (rightmost images in figure 6.3) both have optimizers have minimized the loss to a comparable low value, but in the process produced very different transfer results. Notice that Adam manipulates pixels uniformly at random, which is visible as noise throughout the entire image, while Momentum appears to focus on the areas relevant for the target's style, such as the letter's borders, and especially below the T's arms.

Having a closer look at the classification metrics over the same period of iterations (see figure 6.4), we see that by step 60 both optimization algorithms achieve a similar style similarity score (blue lines). The content similarity score for Adam at step 60, however, is much lower. Since Adam very quickly maximizes style classification (over 90 % by step 10) it is most likely that



(a) Adam at learning rate 0.01



(b) Momentum 0.95 at learning rate 1.0

Figure 6.3: pixel manipulation of transfer image at step 10, 20, 30, 40, 50 and 60 (from left to right)

the content similarity score from that point on drops due to the increasing noise in the transfer image while the algorithm overfits on the style image.

Given the described differences in the results, both qualitative and quantitative, I have opted for Momentum 0.95 with a learning rate of 1.0 as the configuration of the optimization algorithm for all following experiments.

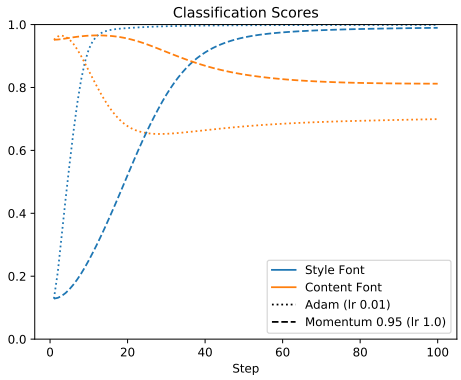
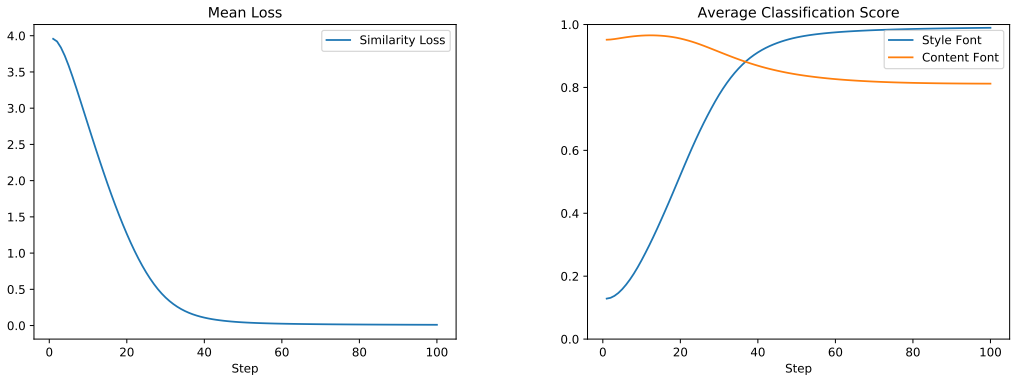


Figure 6.4: comparison of classification scores against the style font (blue lines) and the content font (orange lines) of Adam (dotted lines) and Momentum (dashed lines)

6.2 Basic Approach

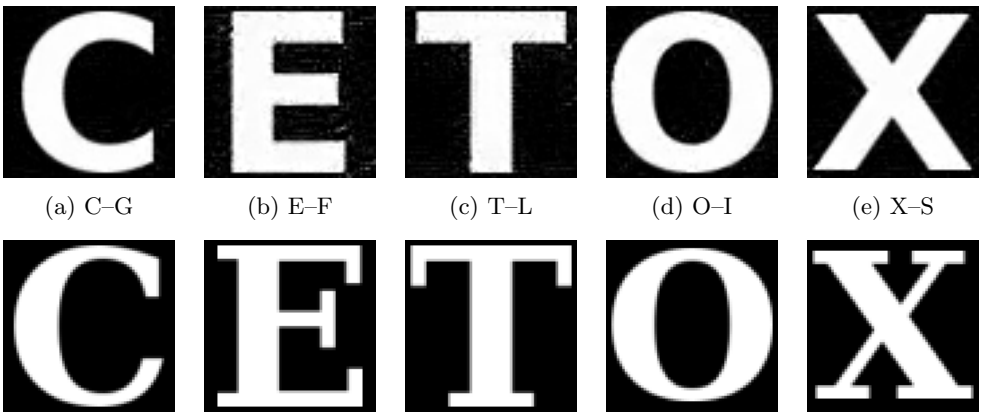
As we have previously seen (in section 6.1) the loss smoothly reaches convergence by step 60, at which point the average style font classification has reached a satisfying accuracy (see blue curve in figure 6.5b).



(a) (b)

Figure 6.5: performance metrics of basic sans-to-serif transfer

Looking at same concrete transfer results (upper row in figure 6.6) it is obvious that the generated glyphs are nowhere near the desired outcome. I have selected both letter combinations that intuitively seemed easy and other than seemed difficult: generate a serif C from a sans C and the style of a serif G, a serif E from sans E and serif F, a T where the style letter is L, a combination O–I and X–S.



(a) C–G (b) E–F (c) T–L (d) O–I (e) X–S

(f) Expected C (g) Expected E (h) Expected T (i) Expected O (j) Expected X

Figure 6.6: transfer results at step 60 of different combinations of content and style glyph in the sans-to-serif transfer and their corresponding expected results

Judging by the metrics only, the assumption is that the classifier is actually convinced that these transfer images have adapted the stylistic characteristics of the style font. While the results are not convincing to humans, the pixels of noise are enough to fit the classifier model's criteria for the style font's stylistic features. The transfer images have been merely manipulated to trigger a similar response as the glyphs of the style font. This phenomenon is commonly known as an adversarial attack, in which images are deliberately manipulated to trick an image classifier. In the more sophisticated attacks the minimal changes necessary to confuse a classifier are not even visible to the human eye [17]. Note how the modifications of C, O and X (figures 6.6f, 6.6i, 6.6j) are barely visible. In the present case, the typographic style classifier has become the victim of it's own creation.

Another aspect in this analysis is important to note. The average content font classification score (orange curve in figure 6.5b), although initially decreasing slightly, has not dropped below 80 % accuracy. That means that the classifier is not only convinced that the transfer results are similar in style to the style font but also to the content font. In section 6.3 I continue to address this issue and the general quality of the transfer results through additional loss terms.

6.3 Loss

6.3.1 Dissimilarity Loss

As a reaction to the previous experiment's results (section 6.2), I introduce the similarity loss's complement as the measure of how much two given glyph images are different in typographic style. Let's call it Dissimilarity Loss. The goal is to reduce the similarity to the content font in the hope to improve the generated image's quality.

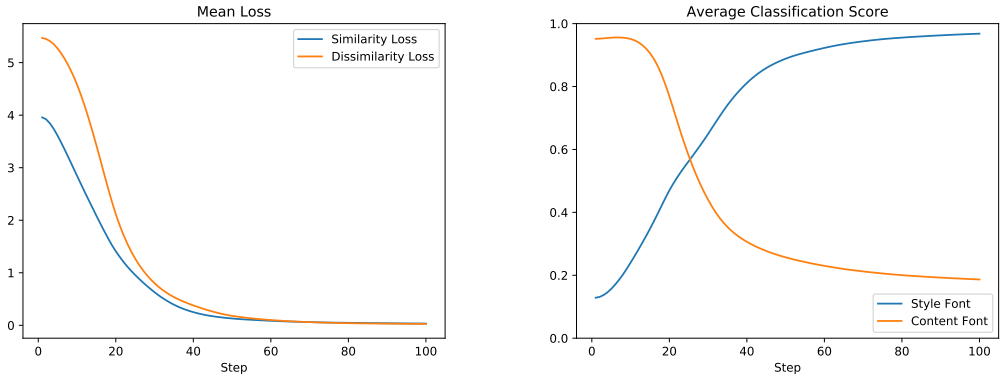
Instead of the style image we feed the classifier model with the content image together with the transfer image and set the optimization target to 0 (false). Consequently, we define this loss's measure as the cross-entropy between the prediction p and the target 0: $L_{\text{dissim}}(p) = L_{\text{BCE}}(p, 0)$

The composite loss term is now defined as the sum of the similarity loss and the dissimilarity loss, where p_s is the output of the classifier M given the style image $\mathbf{s}$ and transfer image $\mathbf{t}$ as input, and p_c the output given the content image $\mathbf{c}$ and transfer image $\mathbf{t}$.

$$L = L_{\text{sim}}(p_s) + L_{\text{dissim}}(p_c) \quad p_s = M(\mathbf{s}, \mathbf{t}) \\ p_c = M(\mathbf{c}, \mathbf{t})$$

Results

Figure 6.7 shows the performance on the composite loss term and the corresponding classification results. We can observe that both losses are simultaneously decreased until smoothly reaching convergence, still by step 60. The average classification scores, while now rising more slowly for the style font, now drops noticeably fast for the content font towards the 20 % accuracy mark. The additional Dissimilarity Loss is successful in achieving the expected quantitative results.



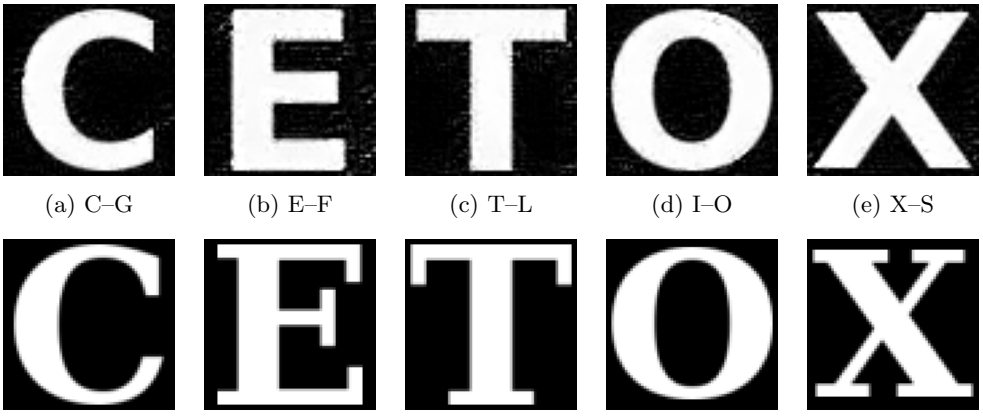
(a) mean Similarity Loss

(b) average classification scores against style font and content font

Figure 6.7: performance metrics of sans-to-serif transfer with composite loss term

However, a quick look at the transfer glyphs reveals that the algorithm is still producing adversarial examples (see figure 6.8). In comparison to the previous results the pixels that are being modified have changed and others have become more prominent. The C–G transfer previously showed almost no visible alterations (fig. 6.6a), whereas now they are significantly more pronounced. The overall impression is that the visible noise has increased. In the next section 6.3.2 I will continue to work on improving the image quality.

But first, I want to take a closer look at the presented transfer results. When we actually classify the transfer images against both the style and the content fonts, we obtain precise measures of only a single generated glyph, whereas we have only been looking at average results over all 676 transfer images. Table 6.1 shows what has not been visible in these average scores. There are two columns for two different results, E-F and T-L, corresponding to the figures 6.8b and 6.8c. For either of the two the probability of similarity in style to all glyphs of the style and the content fonts has been measured. For each of these glyphs the corresponding cell contains a classification score of the transfer image at step 60 and in brackets the change of this score since step 0, when the transfer image was identical to the content image.



(a) C–G (b) E–F (c) T–L (d) I–O (e) X–S

(f) Expected C (g) Expected E (h) Expected T (i) Expected O (j) Expected X

Figure 6.8: Transfer results at step 60 of different combinations of content and style glyph in the sans-to-serif transfer

These results show that for the E-F transfer example the style score has increased as intended by about 97 % to over 99 % accuracy. However, the average content score has only dropped very slightly to 76 %, and remains much higher than we were able to deduce from the average results. But, for this transfer image's glyph of reference in content (E, marked in green in the second column) the score has been decreased significantly by more than 96 % only a little over 3 % accuracy. Almost all other glyphs of the content font (the exceptions being B and R) still mark a very high style similarity to the transfer result.

We can observe the same phenomenon but in its inversion for the T-L transfer example. The only glyphs from the style font the transfer result has a high stylistic similarity to are D and its reference stylistic L. The content score has dropped as intended for all glyphs of the content font (with the exception of K).

This behavior can be understood as some kind of overfitting on the only data the algorithm is given: the respective references in style and in content, only one glyph each. As a reacting to this insight, I am exploring 1–N transfer in section 6.4.

	E-F		T-L	
Char	Style Score	Content Score	Style Score	Content Score
avg	0.9973 (+0.9717)	0.7691 (-0.1949)	0.1504 (+0.0749)	0.0619 (-0.8803)
A	0.9984 (+0.9766)	0.9711 (-0.0136)	0.1792 (+0.1327)	0.0266 (-0.9498)
B	0.9996 (+0.9962)	0.3630 (-0.6197)	0.1153 (-0.0035)	0.0007 (-0.9339)
C	0.9993 (+0.9953)	0.9791 (+0.0281)	0.0660 (-0.0662)	0.0197 (-0.8794)
D	0.9996 (+0.9967)	0.4838 (-0.4808)	0.7746 (+0.7069)	0.0145 (-0.9731)
E	0.9987 (+0.9983)	0.0311 (-0.9660)	0.4745 (+0.4709)	0.0189 (-0.9054)
F	0.9984 (+0.9984)	0.5477 (-0.4403)	0.0086 (+0.0078)	0.0107 (-0.8579)
G	0.9983 (+0.9821)	0.9424 (-0.0502)	0.0742 (-0.2905)	0.0143 (-0.9405)
H	0.9998 (+0.9986)	0.6737 (-0.3223)	0.3564 (+0.3238)	0.0132 (-0.9789)
I	0.9979 (+0.9267)	0.9709 (+0.2079)	0.3113 (+0.2814)	0.3483 (-0.5274)
J	0.9987 (+0.9833)	0.9793 (+0.0235)	0.0258 (+0.0027)	0.1152 (-0.7322)
K	0.9990 (+0.9987)	0.8913 (-0.0987)	0.0544 (+0.0515)	0.5814 (-0.4114)
L	0.9983 (+0.9961)	0.6604 (-0.3184)	0.7601 (+0.7517)	0.0937 (-0.7614)
M	0.9990 (+0.9979)	0.9563 (-0.0387)	0.0842 (+0.0739)	0.0184 (-0.9790)
N	0.9862 (+0.9860)	0.8403 (-0.1492)	0.0124 (+0.0100)	0.0253 (-0.9574)
O	0.9991 (+0.9139)	0.9380 (-0.0319)	0.0713 (-0.2875)	0.0080 (-0.9862)
P	0.9992 (+0.9985)	0.7329 (-0.2587)	0.0048 (-0.0008)	0.0021 (-0.9751)
Q	0.9994 (+0.8494)	0.9962 (+0.0194)	0.0011 (-0.1074)	0.0122 (-0.9571)
R	0.9996 (+0.9972)	0.2136 (-0.7820)	0.1493 (+0.1273)	0.0559 (-0.8981)
S	0.9962 (+0.9852)	0.8044 (-0.1558)	0.0026 (-0.0650)	0.0059 (-0.6811)
T	0.9950 (+0.9824)	0.8021 (-0.0706)	0.1470 (+0.1229)	0.0403 (-0.9454)
U	0.9863 (+0.9860)	0.8760 (-0.1123)	0.0106 (+0.0013)	0.0070 (-0.9726)
V	0.9920 (+0.9912)	0.9616 (+0.0329)	0.0038 (-0.0029)	0.0544 (-0.9322)
W	0.9995 (+0.9974)	0.9674 (-0.0192)	0.0168 (+0.0031)	0.0319 (-0.9017)
X	0.9955 (+0.9950)	0.8904 (-0.1023)	0.1099 (+0.1084)	0.0577 (-0.9376)
Y	0.9981 (+0.9967)	0.7541 (-0.1208)	0.0269 (+0.0228)	0.0139 (-0.9364)
Z	0.9979 (+0.7382)	0.7687 (-0.2275)	0.0683 (-0.4284)	0.0184 (-0.9763)

Table 6.1: Classification scores of the transfer images at step 60 (change since step 0) when compared to the glyphs of both the content and style fonts. Marked in green: the content and style glyphs used as references in the specific transfer.

6.3.2 Total Variation Loss

Total variation denoising is a process that reduces the total variation in a signal. In image processing it is often used for noise removal as it enforces spatial smoothness in an image. Having been pioneered by Rudin, Osher, and Fatemi [18], it has found its use as a regularization term in convolutional neural networks through style transfer and super-resolution applications [19].

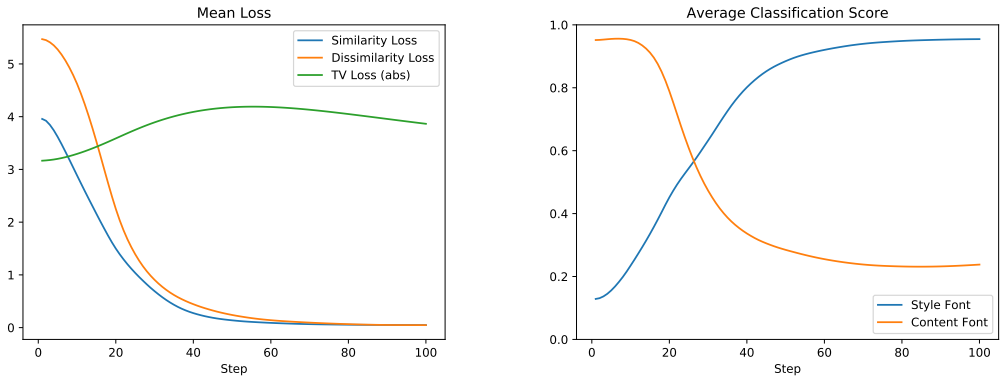
In the given task of regulating the amount of noise in the transfer image, I have tested the regularization of both absolute and squared variation. The total variation loss is calculated on a given image $\mathbf{x}$ as:

$$L_{\mathrm{TV}}(\mathbf{x}) = \sum_{ij} |x_{i,j+1} - x_{ij}| + |x_{i+1,j} - x_{ij}|$$

I have further introduced weighting coefficients for all losses such that the composite loss term is defined as follows. The optimal parameters have been tested and determined as $\alpha = \beta = 1$ and $\gamma = 0.01$.

$$L = \alpha L_{\mathrm{sim}}(p_s) + \beta L_{\mathrm{dissim}}(p_c) + \gamma L_{\mathrm{TV}}(\mathbf{x})$$

Results



(a) (b)

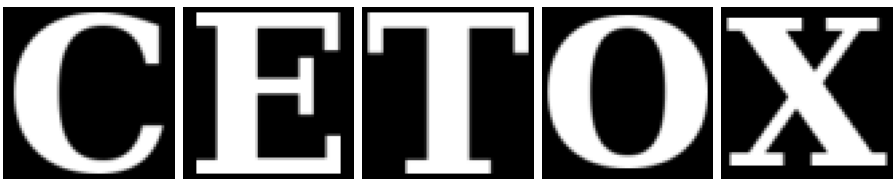
Figure 6.9: Performance of TV loss with a weight of 0.01

With the additional Total Variation Loss (TV Loss) the algorithm needs approximately 20 steps more to reach convergence (figure 6.9). Although the TV Loss is not minimized as the other two,

it does do its job of containing the amount of noise. After rising slightly in the first iterations it reaches its peak at about step 55 and from there on turns to decline again. This apparently goes at the expense of the classification scores. In figure 6.9b both curves have lost slightly in steepness, such that the content font classification score (orange curve) only reaches its low (23 %) 20 steps later than before. In figure 6.10 it is visible how the loss (bottom row) has forced black and white areas to uniform brightness, in comparison to the previous results (top row) which have grey pixels especially visible within the white letters. Setting a constraint on the amount of noise limits the models capability to freely manipulate single pixels to match its internal criteria.

(a) No TV Loss

(b) TV Loss with a weight of 0.01



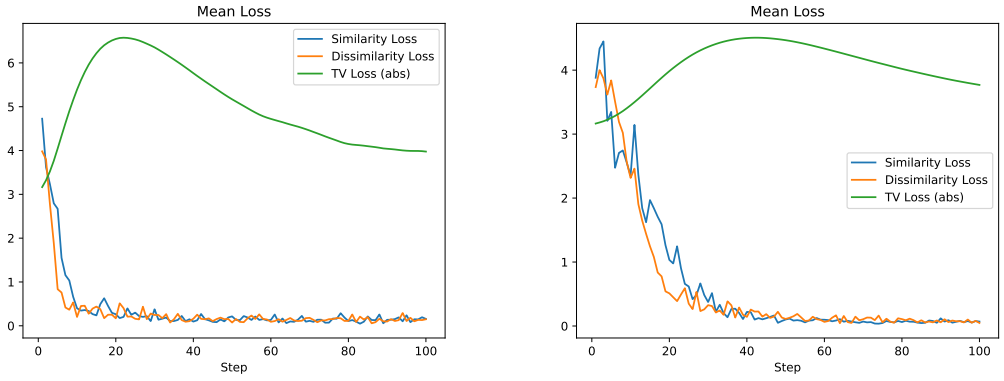
(c) Expected

Figure 6.10: Comparison of transfer results without and with TV Loss, along the expected outcome

6.4 1–N Transfer

Instead of computing the Similarity and Dissimilarity losses always based on the same style and content image, I have adapted the algorithm to select a random glyph from both the style and content fonts at each iteration. This way the Typographic Style Transfer is guided not only by a single reference image, but can infer from the characteristics of the whole font. Let's call this approach 1–N Transfer, whereas the procedure with a single reference image is 1–1 Transfer.

Results



(a) Learning rate 1.0

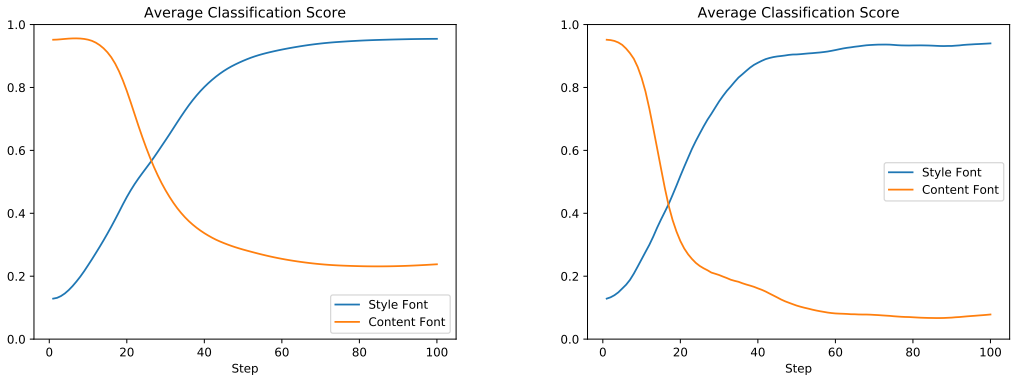
(b) Learning rate 0.1

Figure 6.11: Comparison of loss convergence at different learning rates

Empowered by the newly available information, the algorithm is now notably faster in minimizing the overall loss (see figure 6.11a). But similar to my findings in section 6.1, where I compared the influence of different optimization algorithms, this rushed convergence resulted in excessively noisy transfer images. After lowering the learning rate to 0.1 the optimization goes back to approaching convergence as smoothly as before (figure 6.11a), modifying transfer images more cautiously and reaching convergence around step 60. Although, of course, the loss curves show much higher fluctuations than in the 1–1 Transfer, due to the varying input at each iteration.

Comparing the classification scores of 1–1 and 1–N Transfer (figure 6.12), we can observe that especially the content font classification score drops much faster and well below the 10 % similarity mark.

Table 6.2 (39) shows the same comparison as in the previous experiment: the classification scores of two example transfers against the style and content fonts with the changes from step 0 in brackets. The general improvement compared to the results of the 1–1 Transfer (compare table 6.1) is clearly manifested in the high style scores and low content scores for both examples.



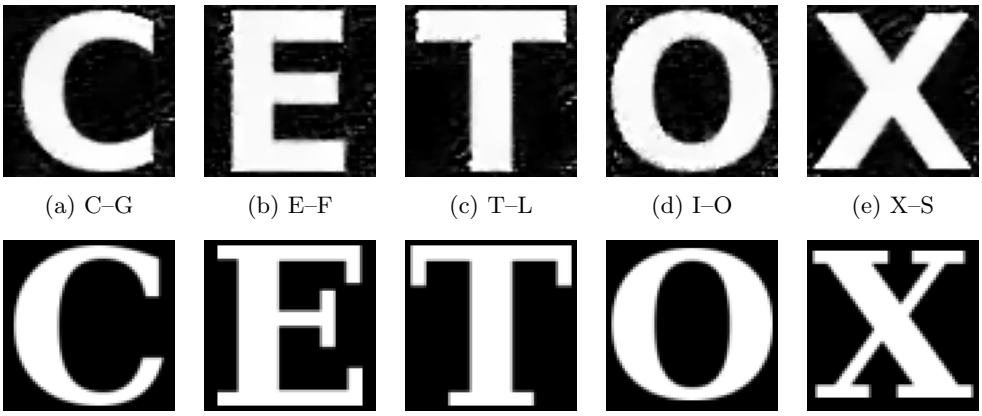
(a) 1-1 Transfer (b) 1-N Transfer

Figure 6.12: Comparison of classification scores of two transfer approaches

There are some remaining outliers like the C in the second column which still reaches 70 % similarity to the E-F transfers, or the Q in the third column which only has 43 % probability of style similarity to the T–L transfer.

Given these detailed measurements, it would be easy to devise a heuristic for the step-by-step selection of reference glyphs during the transfer process. Whichever character from the style font scored the lowest (and vice-versa for the content font) similarity at any given step will be selected as input for the next iteration.

In any case, with the presented strategy it is already much more difficult for the transfer to overfit on the stylistic characteristics of a single reference glyph. Albeit the good quantitative results,



(a) C–G (b) E–F (c) T–L (d) I–O (e) X–S

(f) Expected C (g) Expected E (h) Expected T (i) Expected O (j) Expected X

Figure 6.13: Transfer results at step 60 of different combinations of content and style glyph in the sans-to-serif transfer

the image quality itself has experienced only little improvements (see figure 6.13). I believe, that still it is possible to see a somewhat focused effort of the model to modify specific areas of the transfer image. See for example the underside of the arms of the T (fig. fig:exp-31:results:T-L) where white pixels need to be removed, or the left hand side of the bottom of its stem where a serif would need to be placed. Whether this is a hint of an intention or merely a reflection of an internal representation is hard to tell. And in any case, the model is already convinced that these minor changes are enough to constitute a change in typographic style, which from a human point of view clearly is not the case.

I have used this final setup of composite loss term and random style and content image for experiments in the following two sections 6.4 and 6.5, as well as to compute a final set of results presented in appendix A.

	E-F		T-L	
Char	Style Score	Content Score	Style Score	Content Score
avg	0.9233 (+0.8977)	0.1457 (-0.8183)	0.9164 (+0.8409)	0.0738 (-0.8684)
A	0.9672 (+0.9454)	0.2507 (-0.7340)	0.9170 (+0.8705)	0.0471 (-0.9293)
B	0.9901 (+0.9867)	0.0068 (-0.9759)	0.9933 (+0.8745)	0.0015 (-0.9331)
C	0.9773 (+0.9733)	0.7051 (-0.2459)	0.9190 (+0.7868)	0.0454 (-0.8537)
D	0.9870 (+0.9841)	0.1641 (-0.8005)	0.9987 (+0.9310)	0.0029 (-0.9847)
E	0.9545 (+0.9541)	0.0031 (-0.9940)	0.9975 (+0.9939)	0.0040 (-0.9203)
F	0.9523 (+0.9523)	0.0367 (-0.9513)	0.9903 (+0.9895)	0.0037 (-0.8649)
G	0.9864 (+0.9702)	0.3022 (-0.6904)	0.9281 (+0.5634)	0.0203 (-0.9345)
H	0.9927 (+0.9915)	0.0088 (-0.9872)	0.9988 (+0.9662)	0.0136 (-0.9785)
I	0.9089 (+0.8377)	0.2400 (-0.5230)	0.9763 (+0.9464)	0.5053 (-0.3704)
J	0.9940 (+0.9786)	0.6145 (-0.3413)	0.9334 (+0.9103)	0.2211 (-0.6263)
K	0.9742 (+0.9739)	0.1943 (-0.7957)	0.9959 (+0.9930)	0.1046 (-0.8882)
L	0.9480 (+0.9458)	0.0600 (-0.9188)	0.9966 (+0.9882)	0.0206 (-0.8345)
M	0.6969 (+0.6958)	0.0381 (-0.9569)	0.9741 (+0.9638)	0.1243 (-0.8731)
N	0.7494 (+0.7492)	0.0116 (-0.9779)	0.8677 (+0.8653)	0.0383 (-0.9444)
O	0.9846 (+0.8994)	0.0871 (-0.8828)	0.8082 (+0.4494)	0.0332 (-0.9610)
P	0.9618 (+0.9611)	0.0249 (-0.9667)	0.9843 (+0.9787)	0.0021 (-0.9751)
Q	0.9473 (+0.7973)	0.3402 (-0.6366)	0.4373 (+0.3288)	0.0805 (-0.8888)
R	0.9623 (+0.9599)	0.0114 (-0.9842)	0.9975 (+0.9755)	0.0019 (-0.9521)
S	0.9702 (+0.9592)	0.3041 (-0.6561)	0.5882 (+0.5206)	0.0090 (-0.6780)
T	0.7944 (+0.7818)	0.0417 (-0.8310)	0.9866 (+0.9625)	0.3788 (-0.6069)
U	0.7716 (+0.7713)	0.0212 (-0.9671)	0.9453 (+0.9360)	0.0107 (-0.9689)
V	0.8676 (+0.8668)	0.0656 (-0.8631)	0.8953 (+0.8886)	0.0560 (-0.9306)
W	0.9400 (+0.9379)	0.1228 (-0.8638)	0.9222 (+0.9085)	0.0897 (-0.8439)
X	0.9366 (+0.9361)	0.0492 (-0.9435)	0.9606 (+0.9591)	0.0468 (-0.9485)
Y	0.9414 (+0.9400)	0.0401 (-0.8348)	0.9807 (+0.9766)	0.0184 (-0.9319)
Z	0.8496 (+0.5899)	0.0443 (-0.9519)	0.8338 (+0.3371)	0.0394 (-0.9553)

Table 6.2: Classification scores of the transfer images at step 60 (change since step 0) when compared to the glyphs of both the content and style fonts. Marked in green: the content and style glyphs used as references in the specific transfer.

6.5 Model States

In this previous sections I have first trained a custom classifier and then used the model state in the subsequent style transfer process. But two models are not necessarily equal to each other, and neither are two training states. But what is the difference exactly, and which influence, if any, does it have on the subsequent application?

While this is an open research question and does in fact apply in a broader sense to any use of a model which it was not trained for (such as transfer learning), I will in this section explore the correlation of model state to transfer results for the specific given problem.

For this purpose I have retrained the same network, as described in chapter 4 and saved its state during training at three different epochs at regular intervals (see figure 6.14). At epoch 566 (when the model achieved 97.12 % accuracy), at epoch 1081 (98.08 %) and at epoch 1649 (98.51 %).

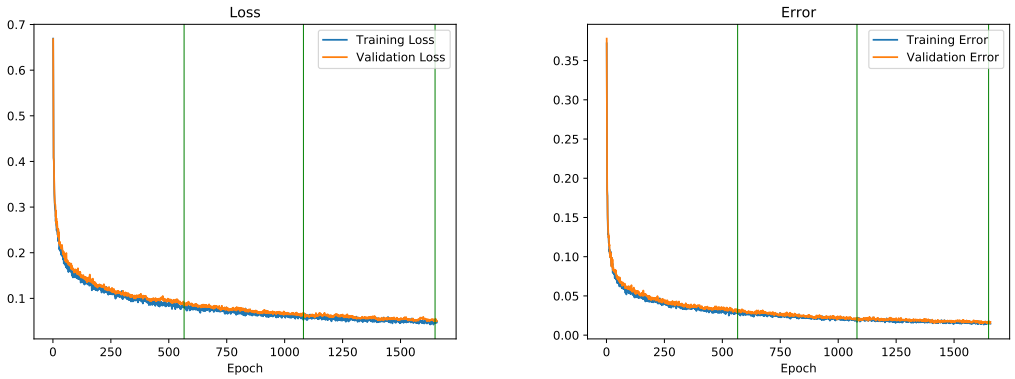
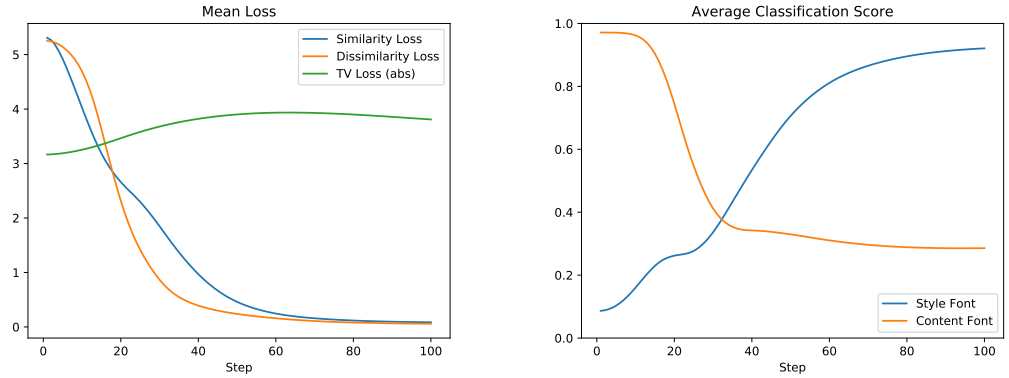


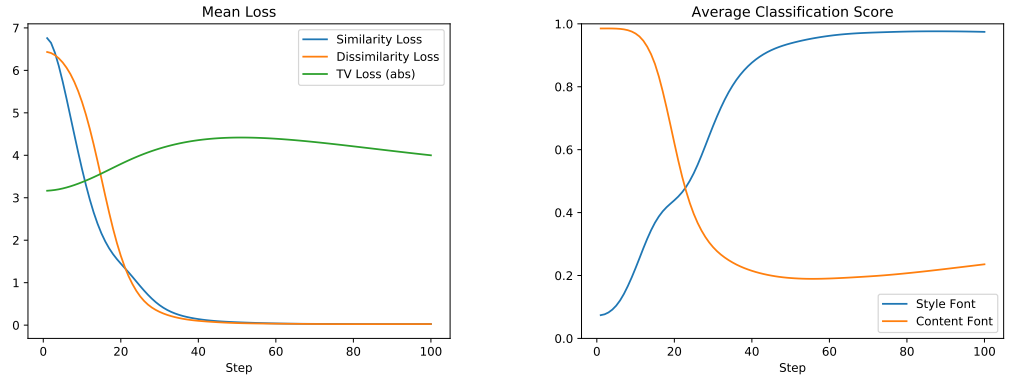
Figure 6.14: Losses and errors measured during training, the saved states are marked in green

In this experiment all transfers have again been computed with the final 1–N Transfer setup as described in section 6.4. Figure 6.15 is an overview of the metrics when performing the same style transfer task with different model states. We can observe that the longer the model has been trained the faster it is in performing the style transfer. With the model state from epoch 566, similar to previous results, losses converge after about 80 steps. With the state of epoch 1081 it does reach convergence already at step 55, and even earlier at about step 40 with the model state from epoch 1649. In accordance to these observations the changes in classification scores follow the same logic. With a growing number of training epochs the curves are increasingly steeper and reach a higher maximum or lower minimum. It is likewise noticeable that the initial values for the Similarity and Dissimilarity Loss increase as well. While both started at a value of about 5.2 with a state from epoch 566, they have values of over 6.4 for epoch 1081 and for epoch 1649 especially the Similarity Loss reaches an initial value of 11. All of the described correlations

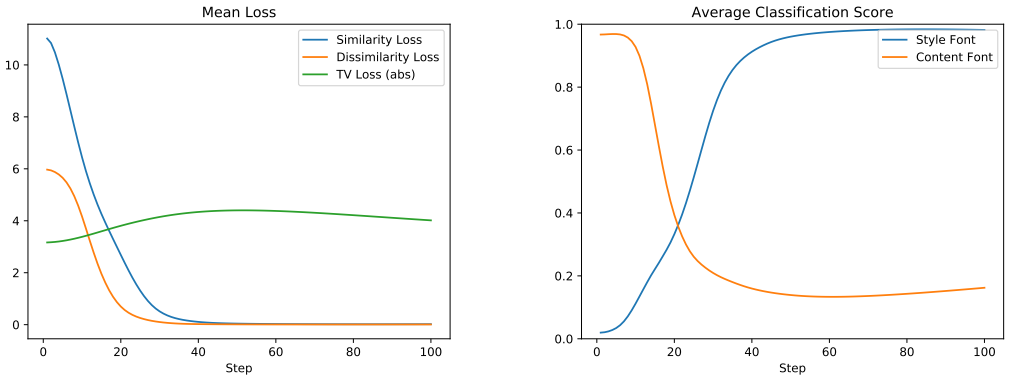
show that the internal organization of the classifier model undergoes immense changes during its training. The TV Loss, however, pretty much behaves the same with all three different model states, since this measure does not depend on the model itself, but only the transfer image.



(a) Model state from epoch 566



(b) Model state from epoch 1081



(c) Model state from epoch 1649

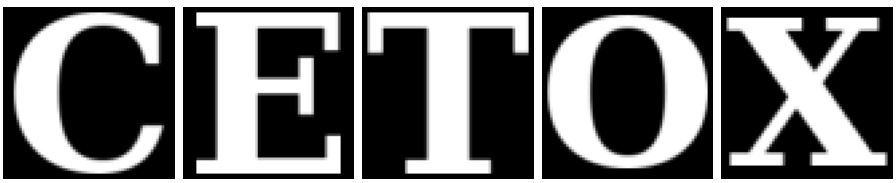
Figure 6.15: Comparison of transfer performance of different model states

A comparison of transfer results displayed in figure 6.16 shows the big differences between model states from different epochs. Note that even the closest state (epoch 566) to the model used in previous experiments (epoch 527, see section 4.3) already produces different results. While in general the effects on the transfer image appear to be stronger the longer the model has been trained, the most pronounced alterations of the transfer images are done with the model state from epoch 1081 (row b). They are, however, also the noisiest and the generated image quality seems to improve by training epoch 1649. At this point it hard to already draw concise conclusion from these few hints. But it would be interesting to see how the transfer results change with even more training of the classifier beyond epoch 1649, since it is not clear if this is only the start of an improvement of internal representation or if the optimal state for a style transfer has already passed and lies within the first 1000 epochs. Equally important would be an investigation into deeper and more powerful network architectures. As I noted before, the performance of the classifier is not yet maxed out and has room for improvement.

(a) Model state from epoch 566, results at step 80

(b) Model state from epoch 1081, results at step 60

(c) Model state from epoch 1649, results at step 50



(d) Expected results

Figure 6.16: Transfer results of different combinations of content and style glyph in the sans-to-serif transfer

Chapter 7

Conclusions

First of all, this typographic style transfer project, in the way that I have approached it, has not produced the expected results. Before going into details of the analysis of failure, let's take a look at what I do have accomplished and learned. Apart, of course from the uncountable small but relevant technical and conceptual lessons.

I have proposed a simple architecture for a convolutional neural network to determine the probability of similar style given two glyph images. I have further successfully trained this network and shown that the resulting model is capable of achieving up to 98 % classification accuracy on the given data set. My results in this task are by a considerable amount higher than the reported 92.1 % accuracy of Baluya's voting ensemble [7], albeit his remarkable effort of testing 60 different network architectures. To be fair, my model has not been tested on the same data and I am aware that he has worked with several pictographic fonts which might pose additional difficulties.

The proposed style transfer methodology does not depend on a pre-trained model and can potentially be adapted to a variety of classification networks and be tested in similar tasks. The insights gained in the process of its development can further be of guidance in comparable problems.

One of those insights is the importance of the choice of the adequate optimization algorithm and its hyperparameters, since it is a central part to the transfer computation and can significantly influence its outcome. Almost of equal importance, as the basis for the given style transfer, is the model itself. Exactly what influence a decision during training later has in the transfer task, and which model state to select remains an open research question. Not only for the present application but also for any other that employs such a model in a transfer task.

Finally, it only remains to say that in the given case none of the above actually has been decisive in the success or failure of the project. It is very hard to look for, compare and then refer to

unsuccessful research since it almost never leaves the laboratory. I have thus not been able to find similar approaches that did not work as expected. It would be exaggerated to claim that artificial neural networks are not suitable for this kind of task, given the other more basic and successful approaches I have collected from the literature. Nevertheless I have encountered a limitation in this specific application. In my intuition, and it would require further investigation to back this up, the internal representation of the custom trained classifier model is not sufficient to be of guidance in the style transfer. The model has not actually captured the notion of style in a typography, but has rather learned to read the right combinations of pixel values and to then base its decision on a number of complex conditions, which is obviously not enough. This inevitably leads us to a conceptual dilemma. What exactly does constitute actual understanding? Is it enough to convincingly appear to have understanding? Can the elusive concept of typographic style even be captured? I certainly now have more questions than I had before.

Gary Marcus has recently summarized his opinions about the limitations of deep learning [20], and states that "deep learning thus far is shallow and has limited capacity for transfer". He goes on to argue that "the real problem lies in misunderstanding what deep learning is, and is not, good for." So again, we do have to be fair. Understanding the difference between style and content in the case of typography, where both are embedded in a character's shape as I laid out in the introduction, is extremely hard. Can we even be sure that humans are able to do it well? Maybe only with the necessary expertise and practice. And can we thus expect a machine to solve this for us only from a load of data? I believe, that there is more to the way we perceive, infer and create than can be modeled statistically, especially in such an artistic domain.

Bibliography

[1] Joshua B Tenenbaum and William T Freeman. "Separating style and content". In: *Advances in neural information processing systems*. 1997, pp. 662–668.

[2] Douglas R Hofstadter. "Meta-Font, Metamathematics, and Metaphysics: Comments on Donald Knuth's" The Concept of a Meta-Font"". In: *Visible Language* 16.4 (1982), p. 309.

[3] Donald E Knuth. "The concept of a meta-font". In: *Visible language* 16.1 (1982), pp. 3–27.

[4] Tomo Miyazaki et al. "Automatic generation of typographic font from a small font subset". In: *arXiv preprint arXiv:1701.05703* (2017).

[5] Huy Quoc Phan, Hongbo Fu, and Antoni B Chan. "Flexyfont: Learning transferring rules for flexible typeface synthesis". In: *Computer Graphics Forum*. Vol. 34. 7. Wiley Online Library. 2015, pp. 245–256.

[6] Paul Upchurch, Noah Snavely, and Kavita Bala. "From A to Z: supervised transfer of style and content using deep neural network generators". In: *arXiv preprint arXiv:1603.02003* (2016).

[7] Shumeet Baluja. "Learning typographic style". In: *arXiv preprint arXiv:1603.04000* (2016).

[8] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "Image style transfer using convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2414–2423.

[9] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[10] Samaneh Azadi et al. "Multi-content gan for few-shot font style transfer". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 11. 2018, p. 13.

[11] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[12] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), p. 533.

[13] Matthew D Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[15] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. “On the convergence of adam and beyond”. In: (2018).

[16] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[17] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. “Adversarial examples in the physical world”. In: *arXiv preprint arXiv:1607.02533* (2016).

[18] Leonid I Rudin, Stanley Osher, and Emad Fatemi. “Nonlinear total variation based noise removal algorithms”. In: *Physica D: nonlinear phenomena* 60.1-4 (1992), pp. 259–268.

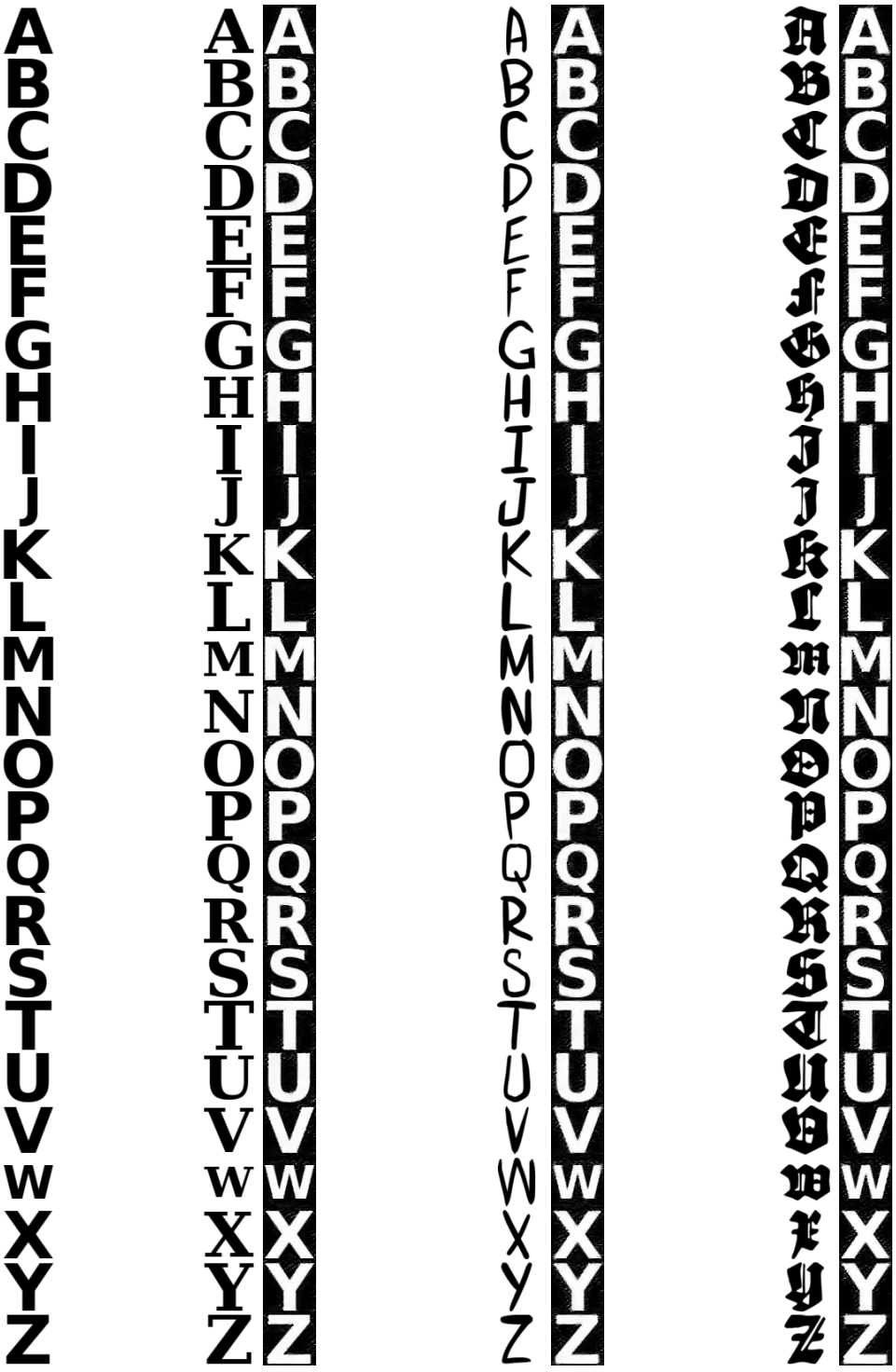
[19] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. “Perceptual losses for real-time style transfer and super-resolution”. In: *European Conference on Computer Vision*. Springer. 2016, pp. 694–711.

[20] Gary Marcus. “Deep learning: A critical appraisal”. In: *arXiv preprint arXiv:1801.00631* (2018).

Appendix A

1–N Transfer Results

For the four fonts from the *test* set presented in section 5.4 this appendix presents the complete character set of transfer results after 60 iterations as computed with the final setup described in section 6.4. The results are grouped by content font (column a) on the following four pages. For each transfer I present the three other style fonts and corresponding results in columns b, c and d.



(a) (b) (c) (d)

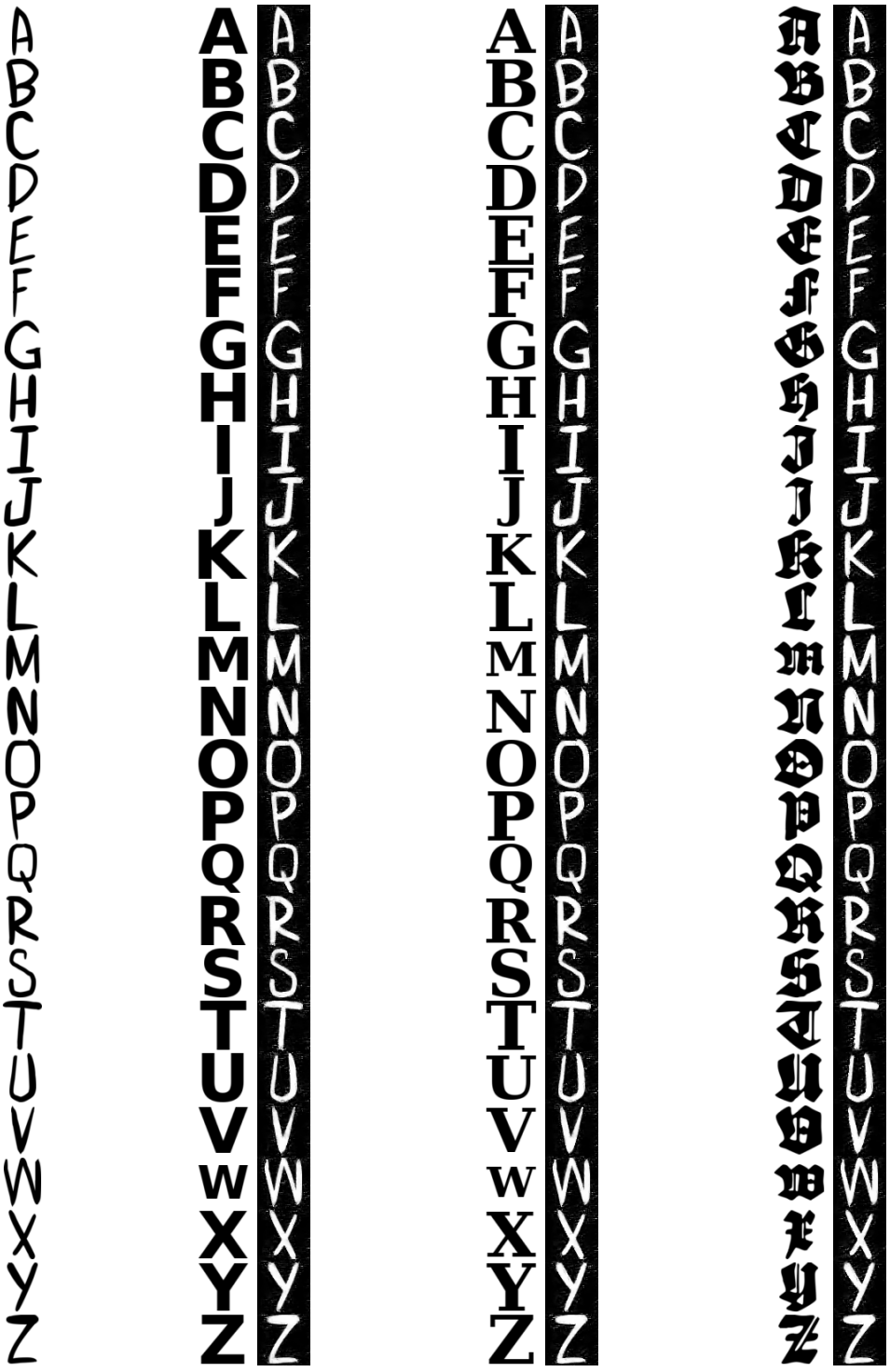
Figure A.1: DejaVu Sans Bold

(a) (b) (c) (d)

Figure A.2: DejaVu Serif Bold

(a) (b) (c) (d)

Figure A.3: Sebaldus Gotisch



(a) (b) (c) (d)

Figure A.4: Miniquest

Appendix B

Glossary

Font A specific instance in weight, slope, width and/or size of a typeface.

Foundry A company that produces typefaces. "Foundry" comes from the era when leaden type was used for printing and movable letters were actually cast from molten metals.

Glyph A concrete instance of a character of a font, and as such a specific design of a letter in the style of a particular typeface.

Glyph image A glyph of an outline font rendered to a pixel grid.

Letterform The shape of a glyph.

True Type Font (TTF) Outline font standard developed by Apple and Microsoft. Mayor competitor to Adobe's PostScript Type 1 font encoding. The glyph data are stored as lists of nodes that connect to shape a glyph's outline.

Typeface A collection of fonts that share a common style and can therefore be considered a font family. A typeface unites fonts of different weights ("Light", "Regular", "Bold", ...), slope ("Italic", "Oblique", "Slanted" ...), widths ("Condensed", "Compressed", "Extended", ...) and sizes ("Caption", "Subhead", "Display", ...). Thus "Helvetica Bold Condensed Italic" is the bold-weighted, condensed-width, italicized font of the typeface "Helvetica".

Open Type Font (OTF) Enhancement of the True Type outline font standard lead by Microsoft in collaboration with Adobe. Combines and extends the True Type and Type 1 (PostScript) technologies.